The Curse of NixOS

JAN 24, 2022

I've used NixOS as the only OS on my laptop for around three years at this point. Installing it has felt sort of like a curse: on the one hand, it's so clearly the only operating system that actually gets how package management should be done. After using it, I can't go back to anything else. On the other hand, it's extremely complicated constantly changing software that requires configuration with the second-worst homegrown config programming language I've ever used[1].

I don't think that NixOS is the future, but I do absolutely think that the *ideas* in it are, so I want to write about what I think it gets right and what it gets wrong, in the hopes that other projects can take note. As such, this post will not assume knowledge of NixOS — if you've used NixOS significantly, there probably isn't anything new in here for you.

The Good

The fundamental thing that NixOS gets right is that software is never installed globally. All packages are stored in a content-addressable store — for instance, my editor is stored in the directory "`/nix/store/frlxim9yz5qx34ap3iaf55caawgdqkip-neovim-0.5.1/`" — the binary, global default configuration, libraries, and everything else included in the vim package exists in that directory. Just downloading that doesn't "install" it, though — there isn't really such a thing as "installation" in the traditional sense. Instead, I can open a shell that has a `$PATH` variable set so that it can see neovim. This is quite simple to do — I can run `nix-shell -p neovim`, and I'll get dropped into a shell that has neovim in the `$PATH`.

Crucially, this doesn't affect any software that doesn't have its `$PATH` changed. This means that it's possible to have as many different versions of the same software package coexisting at the same time as you want, which is impossible with most distributions! You

can have one shell with Python 3.7, another with Python 3.9, and install a different set of libraries on both of them. If you have two different pieces of software that have the same dependency, you don't need to make sure they're compatible with the same version, since each one can use the version of the dependency that it wants to.

Almost all of the good things about NixOS are natural consequences of this single decision.

For instance, once you have this, rollbacks are trivial — since multiple versions of the same software can coexist, rolling back just means changing which version of the software is used by default. As long as you save the information about what versions you used to be on (which is a tiny amount of information), rolling back is essentially just changing some symlinks. Since the kernel is a package like any other, you can have the bootloader remember the list of different versions, and let people boot into previous configurations just by selecting an older version on the boot menu.

This also makes running patched versions of software much simpler — I don't need to worry about fucking up my system by patching something like the Python interpreter, since I know that my patched version will only run when I specifically want it. But at the same time, I can patch the Python interpreter and then have some software running on my system actually use the patched version, since all of this stuff is configured through the same configuration system.

Another advantage to this systems is that it makes zero-downtime deploys significantly simpler, since you can have multiple versions of the same software running at the same time. You don't need to take down the current version of the software before you install the new one, instead you can install the new version of the software, run both at the same time, and then cut over once you're confident that the new version works[2].

Mobile phones and embedded devices have had to build a less general version of this in order to avoid occasionally bricking themselves when they update, in the form of an A/B partitioning scheme. So far, desktop computers, and particularly Linux distributions have largely accepted that occasionally bricking themselves on update is basically fine, but it doesn't have to be this way! Using a NixOS-style system eliminates this problem in a clean, unified manner[3].

One clear reason to believe that this is the future is that language package managers (which are more plentiful and can iterate faster) have largely landed on essentially this solution — `virtualenv`, Poetry, Yarn, Cargo and many others have landed on basically this model. Most use version numbers instead of content-addressable storage, due to the language ecosystems that they're built around, but the fundamentals are the same, and it's pretty clear from looking at trends in package managers that this model tends to be successful.

The Bad

There are essentially two fundamental design mistakes in NixOS that lead to the problems with it.

The first is relatively simple: they developed their own programming language to do configuration, which is not very good and is extremely difficult to learn. The vast majority of people using NixOS do not understand the language, and simply copy/paste example configurations, which mostly works until you need to do something complicated, at which point you're completely high and dry. There seem to be a handful of people with a deep understanding of the language who do most of the infrastructural work, and then a long tail of people with no clue what's going on. This is exacerbated by poor documentation — there are docs for learning Nix as a language, and docs for using NixOS, but the connection between those two things is essentially undocumented. One of the things that's theoretically nice about having everything defined in the Nix language is that it's easily understandable once you learn Nix. Unfortunately, Nix is difficult enough to learn that I couldn't tell you if this is true or not. Nix needs more docs explaining deeply how practical applications of the Nix language actually work. It could also do with less ugly syntax, but I think that ship has sailed.

There are many other minor complaints about NixOS that stem from this — patching packages is theoretically easy, but annoying to figure out how to do in practice, for instance, and configuration tends to have a lot of spooky action-at-a-distance.

The second flaw is that NixOS does not actually provide real isolation. Running `bash -c 'type $0'` will get you `bash is /nix/store/90y23lrznwmkdnczk1dzdsq4m35zj8ww-bash-interactive-5.1-`

`p8/bin/bash` — bash knows that it's running from the Nix store. This means that all software needs to be recompiled to work on NixOS, often with some terrifying hacks involved. It also means that it's impossible to statically know what other packages a given package might depend on. Currently, the way this is implemented is essentially grepping a package for `/nix/store/` to try to figure out what the dependencies are, which is obviously... not great. It also means that binaries that link against `/lib/ld-linux.so.2` or scripts that use `#!/bin/bash` won't work without patching.

Unfortunately, the tools for fixing this are not really there yet. Last fall, I prototyped a Linux distribution trying to combine a nix-store style package repository with overlayfs[4]. Unfortunately, overlayfs becomes very unhappy when you try to overlay too many different paths (with three distinct failure modes, interestingly), which severely limits this approach. I still think that there's a lot of potential here — overlayfs could be fast for arbitrary numbers of paths if that was a design goal — but it's not there yet. This means that trying to build content-addressable store that is transparent to the apps installed in it requires essentially building a container image for every composition of packages (this is the approach that Silverblue takes), which is fundamentally unsatisfying to me.

The advantage to this approach is that you can piggyback off of existing package repositories. One of the main barriers for adoption of new Linux distributions is packaging, but a distribution taking an content addressable store + overlay approach could automatically get all the benefits of NixOS along with all of the packages from Debian, Ubuntu, RedHat, Arch, NixOS, and any other distributions it fancies.

On the whole

NixOS very clearly has the correct way of thinking about dependency management, but is hampered by a few poor technical decisions made long ago. I'm going to keep using it, since I can't stand anything else after having a taste of NixOS, but I'm rooting for something new to rise up and take its place, that learns from the lessons of NixOS and implements its features in a more user-friendly way.

---

1. The worst is, of course, GCL/borgcfg, the (turing complete) configuration language for Google's internal job scheduling software. I used to sit next to the team that was

reimplementing the interpreter for it (since the original interpreter leaked memory all over the place), after performing what they referred to as a "forensic analysis" of the semantics of the language, and they were elated when they finally managed to get 90% of the config files in the monorepo to have the same behaviour as the original interpreter. Truly one of the worst languages I have ever seen. ↩

2. Some people will say that this doesn't matter, because no one does zero-downtime deploys on a single machine anyways. This is bullshit — while it's uncommon, people absolutely do, and a major reason that it's uncommon in the first place is precisely because it's hard! It's possible to get perfectly adequate uptime on a single machine, and a major part of the reason people think it isn't possible is because the tools for doing so have historically been very poor. ↩
3. You will likely still need a tiny bit of manual A/B/R logic for dealing with firmware updates, but putting as much as possible into the general system has a lot of nice properties. ↩
4. If people are interested in the notes from this, let me know. I could polish them up and publish them without too much work. ↩